1983-04-07

M-FORTH 3
Multitasking FORTH implementation for the 6809
USER'S MANUAL

Jens Zander

INTERNAL REPORT
LiTH-ISY-I-0577

SOFTNET USER GROUP

Department of Electrical Engineering
Linköping University
S-581 83 Linköping, Sweden

Linköpings tekniska högskola
Institutionen för Systemteknik
581 83 Linköping

POLITECHNIKA GDAŃSKA

1983-04-07

M-FORTH 3
Multitasking FORTH implementation for the 6809
USER'S MANUAL

Jens Zander

INTERNAL REPORT
LiTH-ISY-I-0577

Abstract

M-FORTH is an implementation of the FORTH-79 standard for the Motorola M 6809 CPU. In addition to the standard word set M-FORTH also supports Multiprocessing. The number of simultaneous tasks is only limited by the amount of memory available. M-FORTH Version 3 is specially designed for the SOFTNET Node computer board (CCØ9). Beside its use in the SOFTNET node, M-FORTH and the CC09 constitute a powerful single board computer with various program development features.

Digitized by the Internet Archive
in 2025 with funding from
Amateur Radio Digital Communications, Grant 151

https://archive.org/details/m-forth-3-6809

M-FORTH 3

Multitasking FORTH implementation for the 6809

Promable Tape version 3.2

J.Zander

Linkoping, December 1982

USERS MANUAL

This manual is provided for the end user. Its content or any part thereof may not be in any form reproducted without the written consent of the author. Use of this program, or any part thereof for any other purpose than single end use by the purchaser is prohibited.

The supplied software is intended for use only as directed by the manual. Use of undocumented features or parameters may cause unpredictable results for which the author cannot assume responsibilty.

Table of Contents

1. Introduction

2. Multitasking

   2.1 Multi-tasking words

      2.1.1 Creating new processes
      2.1.2 Activating processes
      2.1.3 Resource sharing
      2.1.4 External interupts
      2.1.5 Multi-process error handling

   2.2 Implementation

      2.2.1 The process dictionary map
      2.2.2 The scheduler

   2.3. Multitasking programming examples

      2.3.1 Interprocess communication
      2.3.2 External interrupt handling

3. Screen handling and editing

   3.1 Disc Simulation
   3.2 Simple screen editor

4. Using PROM Dictionaries

5. User defined I/O

6. References

   Appendix

A. Hardware requirements

B. MFORTH glossary

C. Error messages

D. Definition memory map

E. Handy reference

1. Introduction

MFORTH version 3 is an implementation of the FORTH-79 standard FORTH for the Motorola 6809 CPU. For reasons of compatibility also the words of the Forth Interest Group (FIG) implementation (FORTH-79) are included. In case of two words from these sets having the same name, the FORTH-79 interpretation has precendence. The FORTH virtual machine and the core dictionary is contained in 4 2716 or 2 2732 EPROMS and will create a stand-alone multi-user/multi-tasking FORTH system. A number of words are added to the standard dictionary for the creation and control of parallel processes. The number of parallel activities or users are only limited by the amount memory and I/O devices available. A real-time clock, RTIME, is included. The system uses a cassette recorder as mass storage. A simple line editor is also added to enable the user to use screens for program storage. The minimal system will only use as little as 4 K RAM storage. Facilities to extend the set of words in PROM are also available. To break indeterminate loops the 6809 NMI hardware interrupt signal is used to warm start the system without destroing any definitions made. A description of the hardware requirements and memory maps are given in Appendix A.

In the sequel we will concentrate on the specific details concerning this implementation. For a description of the FORTH-79 standard words the reader is referred to the FIG-forth glossary[1] and the 79-Standard word set[3]. As quick reference guide, Appendix E can be used. This appendix contains a summary of the most commonly used words and their interpretation. As a primer in FORTH programming, "Starting FORTH" by Leo Brodie of FORTH Inc. is highly recommended. Although he describes a poly-FORTH dialect, most of his words can be used also in this system.

2. Multitasking

2.1 Multitasking words

2.1.1 Creating new processes

The M-FORTH system is supplied with a number of words giving multiprocessing support. With these words it is possible to create new processes. The defining word used for this purpose is PROCESS. It is used in the execution mode in the following manner :

```
200 PROCESS NEWJOB
```

By this sequence a process called NEWJOB is created with 200 bytes of 'private' dictionary space. This process is capable of carrying out some activity completely simultaneous with all other activities (processes) in the system. If the process is to use string or numeric output using PAD, about 160 bytes is the minimum private dictionary size. To check which processes are running in the system at present, the word SYSTAT may be used. Here is an example

```
SYSTAT

  JOB     PRIO  STATUS

0 SYSTEM  100   RUN  *
1 NEWJOB  100   HALT
```

The process 0 called SYSTEM is the default process controlling the users terminal. It is running the usual QUIT-loop, consisting of accepting one line of source code from the terminal and executing/compiling it (see INTERPRET [1]). Further the SYSTAT table indicates the status of the processes. We observe that SYSTEM is in the active or RUN mode. Since we just have created NEWJOB it has not been given anything to execute. Thus NEWJOB is in the non-active or HALT mode. In the rightmost column of the SYSTAT output an '*' indicates the job executing SYSTAT. In this case we typed SYSTAT from the terminal, thus it is the SYSTEM process that produced this output.

Each process is essentially a self-contained FORTH system, although they are all sharing the same basic FORTH code. Thus each process contains its own USER variable table, containing the 'private' status of the process. Beside the FORTH-78 USER variables, there are USER variables dealing especially with the multiprocessing aspects. For instance, one of these new USER variables is ACTIVE, containing a true value if the process is in the RUN mode, and a false value if its in the HALT mode. A process, when created, inherits most of its USER variable values from the creating, 'parent' process. In chapter 3 follows a complete description of these special USER variables.

A process is usually referenced by giving the start address of its USER table. This address is returned by the name of the process,for instance

will return the start address of NEWJOBs USER table.

Processes are essentially of two differrent kinds, terminal handling processes, which are able to communicate with the user via a terminal, or processes, which do not handle a terminal. In this implementation this distinction is used in the error handling routines. (ch 2.1.5) The user variable TERMINAL-JOB indicates of which type the process is. The value 0 (false) indicatesa non-terminal process, and a content not equal to 0 (true) indicates a terminal handling process. The default value for TERMINAL-JOB when creating a new process is 0 (false). For a terminal process, a serial I/O port other than the SYSTEM terminal can be specified. The address of this port can be stored in the user variable IOPORT. The I/O device thus specified will be used by the process for EMIT,KEY and ?TERMINAL. The default value of IOPORT will be inherited from the 'parent' process.

To remove a process from the system, use the word REMOVE. Never FORGET a process or any word 'below' a process in the dictionary, without first having REMOVEd it from the system. Otherwise unpredictable results will be produced.

2.1.2 Activating processes

A process is activated by setting its ACTIVE variable to a non-zero value. However, this should not be done unless the process has been given a word to execute. This can be done in two different ways. The first, and perhaps the simplest, method is to send one line of source code to the process an let the process execute a text interpreter word. Let us assume that we want our NEWJOB process to print the SYSTAT message on terminal. We type

```
NEWJOB SEND SYSTAT
```

The word SEND takes the rest of the command line, in our case the string 'SYSTAT', and puts it into the terminal (or text-) buffer of NEWJOB and starts a text interpreter called DEF-TASK. This interpreter is defined like

```
: DEF-TASK  BEGIN INTERPRET HALT AGAIN ;
```

This word resembles very much to QUIT, and will interpret one line of source code. However, after that DEF-TASK halts the process, i.e puts it in the HALT mode, instead of repeatedly waiting for new lines as QUIT. The output from the example above will thus be

```
0  SYSTEM 100  RUN
1  NEWJOB 100  RUN  *
```

since NEWJOB will be using DEF-TASK to interpret the string 'SYSTAT'. Note that SEND does not compile the text and can thus only be used in

the Execute state, and not in definitions. The other method of starting execution of other processes is by using the word RUN. The syntax is the following

NEWJOB RUN TEST

NEWJOB will here start to execute the previously defined word TEST . If TEST is not found in the dictionary the usual error message is given (Err 0) and NEWJOB remains in the previous mode. The RUN word runs thus only previously compiled words. The RUN word is a compiling (immediate) word and it can thus be used in definitions in the same manner as above. Great care should be exercised when defining words to be executed by the RUN word. In such a definition execution must never reach a top level ';' (or ;S), since after popping the return stack at the top level no sensible execution address will be found by the interpreter. Definitions of this kind should either

1. Never terminate (End-less loop like QUIT)
   or
2. Terminate execution with a HALT (see below)

Otherwise unpredictable results will be generated. A running process is able to halt itself with the word HALT. Caution: Never use the word HALT in the execution mode of SYSTEM. This results in halting the SYSTEM process and with that the interpreter. When no other process is active, however, the SYSTEM process will be active, no matter the state of the ACTIVE flag. This is a safety precaution to prevent the user of loosing control of the system. To halt some other process we can use the word STOP.

NEWJOB STOP

will halt the NEWJOB process. Processes halted in this manner may be restarted by using the word CONTINUE.

NEWJOB CONTINUE

puts NEWJOB in the RUN mode again. Caution: Never CONTINUE a process that has come to halt of its own or as result of an error condition. These processes should be reset by giving them something to execute by RUN or SEND.

The numbers printed by the SYSTAT command for each process are the relative priorities of the processes. A process having a relative priority three times larger than another process will get three times more time slots by the scheduler, and thus execute three times faster than the other jobs. The magnitude of the priority number is unimportant, time slices are shared in proportion to the quotient of the numbers. The default priority is 100. The priority numbers have to be positive and their sum should not exeed single precision range ($0<p<32767$) to ensure proper operation. The priority of a process is altered by the SET-PRIO word in the following manner:

200 NEWJOB SET-PRIO

This will change the realtive priority of NEWJOB to 200.

2.1.3 Resource sharing

When running several different processes the sharing of common memory areas or I/O devices are problems of vital importance. Since processes are running unaware of the other processes in the system, measures have to be take to exclude unintended interaction between processes. This can be achieved with the common concept of semaphores [2]. In the M-FORTH system these are implemented as common VARIABLEs together with the two words WAIT and SIGNAL. With semaphores it is possible to create program sequences which are never executed by two processes simultaneously even though the program code itself is shared between them. The use is as follows. First we create some semaphore variable called, say SEMA.

```
VARIABLE SEMA    1 SEMA !
```

We can now create a sequence of words in a definitions that are non-sharable by using WAIT and SIGNAL as

```
: TEST  SEMA WAIT <non-sharable sequence> SEMA SIGNAL ;
```

WAIT will check if SEMA is not zero. If this is the case execution will proceed and SEMA is decremented by one. In our case, if another process enters TEST and executes WAIT it will find SEMA to be zero. WAIT will now suspend execution until SEMA is found to have a non-zero value. SIGNAL increments the semaphore by one. Thus, the initial value of the semaphore variable determines how many processes are allowed to have simultaneous access to the "non-sharable" sequence. In our example this initial value was one, allowing only one process at the time to run the word TEST. Both SIGNAL and WAIT are non-sharable themselves to ensure proper operation. Both words should, and WAIT must only be used in definitions. For other examples the reader is referred to [2].

2.1.4 External interrupts

This multitasking version of FORTH supports the handling of a hardware interrupt request, the 6809 FIRQ signal. This is done in much the same manner as handling multiple simultaneous processes. The interrupt is handled by a special process, the interrupt process or interrupt handler. To create such a process we use the word INTPROCESS in the same way we used PROCESS, i.e.

```
200 INTPROCESS IHANDLER
```

will create an interrupt handling process IHANDLER with 200 bytes of private dictionary space. The SYSTAT word will now reveal the following:

```
0  SYSTEM   100  RUN *
IP IHANDLER      HALT
```

The interrupt handling process will not be CONTINUEd by the scheduler unless an external FIRQ signal is sensed by the CPU. Then, the interrupt process will immediately activated, that is to say the next time the inner interpreter is called. This process will keep on running until the next timer interrupt, or scheduler call (SWITCH, cf 2.2.2) In all other ways the interrupt handler exhibits the same benaviour as all other processes. The words RUN, SEND, CONTINUE STOP etc. can all be used to control the process as described in this chapter. All words used by interrupt handler should begin each call executing the word KEEP, which reserves the whole next time slice for the process. Otherwise, the process could be interrupted by a timer interrupt after a duration less than one time slice (here 20 ms). All service handlers should end with SWITCH (endless loops) or HALT (single request service).

For highly time critical applications, it is also possible to use then hardware Fast Interupt Request itself to enter some machine code sequence. This is done by writing the address of the machine code handler in the variable FIRQ. At the address written should be executable 6809 code ending with a RTI instruction. This method should only be used if the procedure above fails due to real time failure, and is not recommended for the novice FORTH programmer.

2.1.5 Multiprocess error handling

In the standard FORTH system, all error conditions are handled by the word ERROR. This word types the appropriate error message and then executes QUIT, thereby restarting the normal text interpreter. This is not very suitable for terminalless processes, since, after executing ERROR they would be competing for terminal input with SYSTEM. Instead the error handler checks the user variable TERMINAL-JOB. If this variable is true the ordinary QUIT is executed. If, however, the error should be caused by any other process, a message containing the origin of the error condition is typed on the terminal of the process which created the erronous terminal-less job. A list of all possible error messages and their interpretation is found in appendix C. Following that message the error causing process is halted.
Caution: A process halted by ERROR should never be CONTINUED, but reset by using SEND or RUN.
For user defined and/or process specific error handling the user variables WARNING and >ABORT are used. If WARNING contains a 0 (default) ERROR will issue an error message just as above. If the contents of WARNING is negative, however, ERROR will execute the word (pfa) which is stored in >ABORT. The default word in this case is ABORT.

2.2 Implementation

2.2.1 The process dictionary map

When a new process is created by the word PROCESS, space is allocated in the dictionary at HERE for the USER table, stacks, terminal buffer and private dictionary of the process. The memory map of a new process is shown in fig 1.

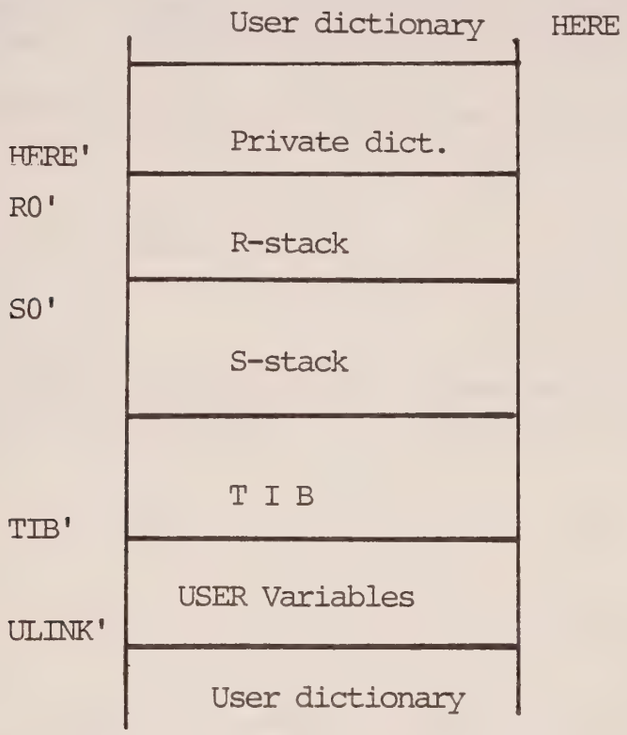


Fig. 1 The process memory map

The USER variable table contains in addition to the standard FORTH USER variables, some special purpose variables. These are listed in table 1 below. These variable are used to save the status of the FORTH-machine when the scheduler switches processes. The status of this machine consist of the R & S - stack pointers (6809 Registers S & U) and the IP register (6809 register Y).

```
0'      ULINK   Link to next process
2'      ACTIVE  Process status ( 0=HALT 1=RUN )
4'      RSTACK  Current R-stack pointer
6'      DSTACK  Current S-stack pointer
8'      IP      Current IP ( Instruction pointer )
        -       -       -       -
        Standard FORTH USER variables
        -       -       -       -
58'     IOPORT  Current ACIA address used for terminal I/O
60'     PRIORITY Relative priority of process
62'     CPRIO   Momentaneous priority of process
64'     TERMINAL-JOB  Terminal job flag
        -       -       -       -
        Standard FORTH USER variables
        -       -       -       -
```

Table 1

The addresses shown in Table 1 are subject to change. To check which actual relative addresses your system is using, use ' <uservar> ? .

2.2.2 The Scheduler

The switching between processes occurs in the inner interpreter of the FORTH system (at NEXT) synchronized to an external timer interrupt. The length of each of the time slices allotted to each process is fixed by the interrupt frequency of the interrupt timer. The present system contains an IRQ timer routine which sets a memory location called IFLAG to a non-zero value each timer interrupt. This flag is sensed by the FORTH machine (NEXT) between every two FORTH words. Is the flag sensed to be non-zero the scheduler is called and processes are switched. In the same way, external interrupts are flagged in the memory location EIFLAG. The 6809 code for the NEXT loop and the scheduler (SWITCH) are shown in listing 1.

```
NEXT    LDX ,Y++        Load and increment IP
        TST IFLAG       Check for timer interrupt
        BNE SWITCH
NEXT3   JMP [ ,X]       Goto next word

SWITCH  LDX UP          Get pointer to current user table
        STS RSTACK,X    Save status
        STU DSTACK,X
        STY IP,X
```

```
        TST EIFLAG      Check for external interrupt
        BNE SYSIRQ
SWTCH1  LDX  SYSTEM     Start with the default process
        TFR X,Y         Save that address
        LDD CPRIO,X     Get current priority of SYSTEM job
        ADDD XPRIO,X    Calculate current priority
        STD CPRIO,X     Update current prio of SYSTEM
        STD MAXPRI      Use as initial value of max cur prio
NXPRO   LDX ULINK,X     Get next process in chain
        BEQ ELIST       End of process list ?
        LDD ACTIVE,X    Get activity status
        BEQ NXPRO       If not active get next process
        LDD CPRIO,X     Calculate current prio of process
        ADDD XPRIO,X
        STD CPRIO,X     Save it
        CMPD MAXPRI     Compare with maximum current priority
        BLE NXPRO       If not greater, get next process.
        STD MAXPRI      Largest priority so far, save it
        TFR X,Y         And save address of process.
        BRA NXPRO
ELIST   STY UP          Load new process
        LDD  0
        STD CPRIO,Y     Clear current priority of loaded job
        LDS RSTACK,Y    Load process status
        LDU DSTACK,Y
        LDY IP,Y
        LDX -2,Y        Restore W-register
        CLR IFLAG       Reset interrupt flag
        BRA NEXT3       Continue execution

SYSIRQ  CLR EIFLAG      Reset external interrupt flag
        LDY INTVEC      Get address of interrupt handler
        BEQ SWTCH1      If no handler, ignore interrupt
        LDD ACTIVE,Y    Check if active
        BEQ SWTCH1      If not, ignore interrupt
        BRA ELIST       Valid interrupt, load handler
```

Listing 1

As can be seen from the listing, the scheduler uses equally long time slices. The process tables are forming a linked list. Each time the scheduler (SWITCH) is called, it will calculate the current or momentaneous priority of all processes. When a process is assigned a time slice, its current priority is reset to zero. To calculate the current priorities, the relative priorities (XPRIO) are added to the current priorities. The next time slice is assigned to the process with the maximum current priority.The address of the next process in the process chain is found in the ULINK USER variable. The end of list is marked by a zero in ULINK.

The scheduler may also be called by a user word if, for example, all work is done by some word, it may surrender the remaining part of its time slice to some other process. This is done by SWITCH , which in fact just sets the interrupt flag IFLAG, thereby causing the

scheduler to be invoked. The word KEEP has a similar function. It sets the external interrupt flag, EIFLAG. At the next scheduler call, either by SWITCH or a timer interrupt the Interrupt process (if any) will be invoked. This can be used in the interupt process itself, to reserve the next time slice. Another application is to invoke the interrupt process from software creating a kind of software interrupt. A suitable time slice duration is 10-30 ms, which is fairly long to reduce overhead time from scheduling. As an example, a system using 20 ms time slices, will have less than 3% scheduling overhead with 4 active processes.

Great care should be exercised when deleting process-tables with FORGET. A process to be deleted from the dictionary should always first be REMOVEd from the process chain. Unpredictable results will otherwise be generated when the freed dictionary is reused for other purposes.

2.3 Multitasking programmming examples

2.3.1 Interprocess communication

In the following we will give a simple example on multiprogramming. We will use the very useful concept of semaphores for interprocess communications and the system time-saving word SWITCH. In this example we want to execute the word TASK-1 and then the word TASK-2 after a real-time delay. Here, we implement this function with a semaphore and two processes, one which does the actual job, and another keeping record of the real time. We start by defining a semaphore for the interprocess communication:

```
VARIABLE SEMAPHORE  0 SEMAPHORE !
```

The semaphore has an initial value of zero , and any process executing a WAIT on it will be suspended until some other process executes a SIGNAL. We create and activate the actual process doing the actual job, the TASK-1 and TASK-2 words:

```
200 PROCESS JOB1

: DO-JOBS  TASK-1 SEMAPHORE WAIT TASK-2 HALT ;

JOB1 RUN DO-JOBS
```

Execution will now start with TASK-1 and will finnaly be suspended when reaching the WAIT word. We define some variables to keep track of the real time:

```
VARIABLE ASEC

: ?TIME ASEC @ RTIME @ = ;
```

The variable ASEC will contain the actual start time of the suspended process (in seconds). The system variable RTIME contains the real time , also in seconds. ?TIME will, of course, return a 1 (true value) on the stack when the start time is reached. We are now ready to create and define the other time keeping process.

```
200 PROCESS TIMER

: TIME-CHECK
    BEGIN  ?TIME
        IF SEMAPHORE SIGNAL HALT
        ELSE SWITCH
        ENDIF
    AGAIN ;
```

The word TIME-CHECK is an endless loop checking the real time with ?TIME. When ?TIME returns a true value, a SIGNAL is executed which will release the suspended DO-JOB word. Otherwise we just keep on checkin the real time. The word SWITCH is very useful here. It has no effect in this process, stacks are kept intact. However, it signals to the scheduler that it has no longer use for the rest of its time slice, and is willing to let some other process have that amount of time. This is very reasonable in this situation, the real time clock will not change its registers for at least one second, so there is no use in checking it that frequently. Finnally, we set the start time variable and start the TIMER process:

```
: SET-TIMER SWAP 60 * + SWAP 3600 * + ASEC !
        0 RTIME ! TIMER RUN TIME-CHECK ;

1 25 00 SET-TIMER
```

The timer is now activated, and DO-JOB will continue with TASK-2 after 1 hour and 25 minutes.

2.3.2 External interrupt handling

The use of hardware interrupts can be very useful when some hardware device requires immediate service, independently of all other processes in the system. To create such a process we use the word INTPROCESS in the same way as PROCESS. A simple example is the one below, in which the process INTHANDLER emits a ASCII Bell (7) to the terminal each time a FIRQ is activated. The user has to provided this signal himself by setting the external devices to generate this hardware signal. KEEP will here ensure the interrupt handler of getting at least one slice duration before being interrupted by the scheduler.

```
200 INTPROCESS INTHANDLER

: INTSERVICE  BEGIN
                KEEP 7 EMIT SWITCH
              AGAIN ;

INTPROCESS RUN INTSERVICE
```

External interrupt programming example

3. Disc Simulation and Editing

3.1 Disc Simulation

The MFORTH 3.nn systems simulate the standard FORTH disc by the use of Tape mass storage. The blocks are stored in the memory starting with Block 4 at adress $800. This corresponds to Screen 1, since the system uses 4 Blocks/screen. The number of screens which can be used is limited only by the upper limit of the RAM. The memory is shared between the Blocks and the dictionary, which starts at address 0. In applications demanding a large dictionary, some of the Block memory can also be used. This is simply done by not using the number of lowermost blocks or screens corresponding to the amount of memory cells needed for that application. One block contains 256 bytes, thus one screen uses 1 K of memory. When compiling large programs the dictionary will grow into the screen area. No conflicts will occur if the the lowermost screen are not used.

The user is able to save his screens or blocks by the use of a simple tape interface. The hardware used for this interface is described in appendix A. The bitrate used for data storage is 1200 bps, permitting the use of fairly low quality audio tape systems. Make sure of having a program version corresponding to your system clock-frequency (1 MHz or 2MHz), since the bit- rate otherwise will be to high or to low. Using a to high bitrate migth cause problems with bit errors. If you are using a high quality audio tape system, it migth be preferable to use the 1 MHz program version with a 2 MHz system, thus using a bitrate of 2400 bps. The user saves his screens by typing

<scr nr> SAVE

which will dump the specified screen to the tape device. A leader of zeros for about 10 seconds, will preceed the screen on the tape for synchronization. The word GET will restore the screen from tape. No argument has to be given to GET, since the screen number is written onto the tape by SAVE. In case of some checksum error while reading the screen, an error message 'Error 9' will be issued. Try reading the screen again with a different volume setting of your tape recorder.

SAVE replaces the standard word SAVE-BUFFER (or FLUSH). Since the user has to handle the disc simulator himself the standard words BUFFER and UPDATE have no meaning and are not supplied in this version.

3.2 Editing

The system incorporates a simple line editor for text handling on screens. The commands are basically the same as the commands of the FIG editor. Since The use of vocabularies is not possible in the PROM area of the dictionary. Thus, standard names could not be used.

All editor commands refer to the current screen, the number of which is stored in the USER variable SCR. Most commands use the line buffer, referenced by PAD, which is located at HERE+64. This buffer is also used as scratch for numeric output conversions. The contents of the line buffer can be printed with the word .PAD . The following editor commands (words) can be used:

n CLEAR	Ereases (fill with blanks) scr n. Makes scr n current screen.
n DL	Delete line n and hold content at PAD
n EL	Erease (fill with blanks) line n
n HL	Hold line n at PAD without removing it.
n IL	Insert line from PAD before line n.
L	List current screen.
n LIST	List screen n and make it current screen.
n PL <text>	Insert text on line n ereasing previous content.
n RL	Replace line n with content of PAD.
n SL	Make new line n, moving old text including the old line n down one line.
c TEXT <text>	Put <text> at PAD until c is encountered. If the byte c is not encountered the whole line is moved to PAD.
.PAD	Type content of PAD buffer

4. Using PROM dictionaries

After developing some application in FORTH it is often of great use, beeing able to store the word set permanently in PROMS for immediate use at power up. There are basically two ways of doing this. The perhaps simplest, but not most efficient way is to copy the source code of some screen into PROM. When the system is powered up, a LOAD is performed on those screens. However simple, this method has some drawbacks. Firstly, even though FORTH source code is compact, large dictionaries will consume a considerable amount of PROM space, and secondly, LOADing the source code screens will take some time. The advantage of this method is, that documentation of the words will be present in the system at all times.

Another metnod is to program only the compiled code in the PROM. This can be down by the following procedure:

1. Install RAM in the area you later want to place your PROM.

2. Make a COLD start of the system and set the user variable DP to point at PROM-address + 2 saving the old value of DP.

3. LOAD your program.

4. Store the value returned by LATEST at PROM address.

5. Reset DP to its previous position and use some program of your own to dump the contents of the PROM into some PROM programmer.

6. Replace the RAM with the PROM and use the word APPLICATION to incorporate your PROM dictionary contents in the system dictionary. The word application takes the PROM-address as parameter off the stack.

Multiple uses of APPLICATION will delete the previous incorporated PROM dictionary. A few hints when using PROM dictionaries are useful. Never use the VARIABLE word in PROM, since the contents will be fixed, thus virtually resulting in the creation of a constant. Instead, use the word CONSTANT to specify a reference into some RAM area, thus giving the same effect as a VARIABLE. Also VOCABULARY uses in-line R/W storage, and will thus not be permitted in PROM. All word have to belong to the (basic) FORTH vocabulary.

Several methods to transfer the RAM content to PROM are availiable. Perhaps the simplest method is to convert the binary memory content to some ASCII format, for instance the common S1-format, and transfer it to some computer system with PROM programming facilities via the terminal line. An example of a function to do this is shown is shown below (fig. 2).

```
 0 ( S1-Code Formatter J.Z. 82-08-13 )
 1  FORTH DEFINITIONS HEX       0 VARIABLE CHKSUM
 2  0 VARIABLE ADDR    0 VARIABLE FINAL  20 CONSTANT BLEN
 3
 4  : B. 0 <# # # #> TYPE ;  : A. 0 <# # # # # #> TYPE ;
 5
 6  : S1LINE CR  ." S1" BLEN 3 + DUP B. CHKSUM ! ADDR @ DUP  A.
 7           DUP  100 /MOD + CHKSUM +!
 8           BLEN 0 DO DUP C@ DUP CHKSUM +! B. 1+ LOOP
 9            ADDR ! CHKSUM @ FF XOR  FF AND B. ;
10
11  : S1CODE  FINAL !  ADDR ! BASE @ HEX
12            BEGIN
13              ADDR @ FINAL @ <  WHILE  S1LINE
14            REPEAT CR  BASE ! ;
15   DECIMAL ;S
```

Fig. 2

The function S1CODE used as <startadress> <endaddress> S1CODE dumps the memory content between <startaddress> and <endaddress> on the screen. The block length used is 32 bytes. The block length is determined by BLEN, and can be set to any value.

5. User definable I/O

The M-Forth 3.1 system enables the user redefine the standard terminal I/O word KEY,EMIT and ?TERMINAL to fit any I/O device of his choice. This is possible by employing vectored execution of these word. This is done in the following manner. The words KEY,EMIT and ?TERMINAL are not executed directly, the interpreter reads the content of the USER variables >KEY,>EMIT and >?TERM respectivly, and executes the words whose PFA address is found. The user defines his owm I/O routines in the following manner. First he defines some words which are to replace KEY,EMIT and/or ?TERMINAL. Then he stores (installes) the PFA address of these words (found by ') in the USER variables >KEY,>EMIT and >?TERM. Since these variables are USER variables these new I/O words are only valid in the process in which they are installed, or any daugther processes of that process. The new words have to have the same effect on the stack as the default I/O words. The default I/O words can also be reinstalled, they are called (KEY),(EMIT) and (?TERMINAL). A small example will clarify this procedure.

Suppose we want to alter the word KEY in such a manner that besides a character it will also return the time it hat to wait for the character to be entered in seconds. Since KEY only returns one value on the stack, our new word has to pass this new parameter via some (global) variable. Thus we define:

```
VARIABLE KEYTIME

: NEWKEY  RTIME @ (KEY) RTIME @ SWAP -
          KEYTIME ! ;
' NEWKEY   >KEY  !
```

This word will be used by all words using the word key, thus of course also the interpreter itself (actually the word EXPECT or QUERY). Note the difference compared to just redefining KEY in the usual manner with a colon definition !. To reinstall the default KEY handler, we just type ' (KEY) NEWKEY ! . Another, a little bit more complex example, using interprocess communication is given below. Here we supply the system with full type-ahead and trapping of C for aborting looping functions. We use a background process, TAPROC running the word (TAH) which puts the incoming characters in the vector TABUFF. Note the use of the vector variables >KEY and >?TERM. The words TAKEY and TATERM take their original place. Another useful word is SELF, which allows a recursive call of the word which is currently defined. In our case, the word SELF compiles the word (TAH) since it is used in the definition of (TAH). ADDP simply adds 1 modulo 50 to the variable at the address at the of the stack.

```
200 PROCESS TAPROC  VARIABLE KEYSEM
VARIABLE RP VARIABLE WP  0 KEYSEM !
CREATE  TABUFF   50  ALLOT
: ADDP DUP @ 1+ 50 MOD SWAP ! ;
: SELF  LATEST PFA CFA , ;  IMMEDIATE
: (TAH)  RP!  SP!  0 RP ! 0 WP !
        BEGIN    BEGIN
                        SWITCH (?TERMINAL)
                 UNTIL  (KEY)
                 DUP 3 = IF SYSTEM RUN QUIT SELF THEN
                 WP @  TABUFF + C!
                 WP ADDP  KEYSEM SIGNAL
        AGAIN ;
: TAKEY  KEYSEM WAIT  RP @ TABUFF + C@  RP ADDP  ;
: TATERM KEYSEM @ ;
' TAKEY >KEY !  ' TATERM >?TERM !  TAPROC RUN (TAH)
```

6. References

[1] FIG-FORTH Glossary, Forth Interest Group,
P.O. Box 1105, San Carlos, CA 94070, USA

[2] Lister,A.M : Principles of Operating systems

[3] FORTH-79 Standard, Forth Interest Group.

APPENDIX A.

Hardware requirements

Standard system memory map

$0 - $7FF	Dictionary RAM	
$800 - $BFFF	Dictionary and Screen RAM/PROM	
$C000	ACIA (6850) or (6551)	(1)
$C200	Watchdog Trig	
$C400	Watchdog On/Off	
$C600	VIA (6522)	(2)
$C800	Baudrate generator 5016	(1)
$D000 - $DFFF	Optional PROM (reserved for future use)	
$E000 - $FFFF	FORTH core dictionary PROM	

NOTES: (1) Check version if system uses the 6551 ACIA with built-in buadrate generator, or the 6850 with the COM-5016 baudrate generator. In the latter case the 6850 may also be used with some fixed baudrate generator (Clock X16), ommiting the 5016.

(2) A VIA is used as timer to generate a 50 Hz IRQ interupt signal. An external 50 Hz interupt may be used instead. In some systems a 50 Hz signal for testing purposes is availabke at the VIA PB7 pin.

The system RESET signal will cause a cold start of the system, just as COLD, clearing all previous definitions. The blocks (and screens) are, however left untouched. The system NMI signal causes a warm start (ABORT), clears the process list, but leaves the dictionary untouched. Processes thus unlinked from the process chain can be restored by LINK-IN. The scheduler triggers the watchdog every time it is called, thereby preventing it from generating a NMI signal.

The Cassette interface.

The cassette interface is a software controlled PSK modulation system with a standard bitrate of 1200 bps, corresponding to a byte transfer rate of 120 bytes/second. The hardware requirements are shown in fig 3. The VIA I/O-pin PA0 is used as input and PA1 is used for output. The output LP filter has to convert the harsh TTL level VIA output to a signal suitable for recording. The input amplifier converts the weak output from the recorder to the VIA TTL-level. For best results use a high quality cassette recorder and carefully adjust input and output levels to minimize the bit error probability.

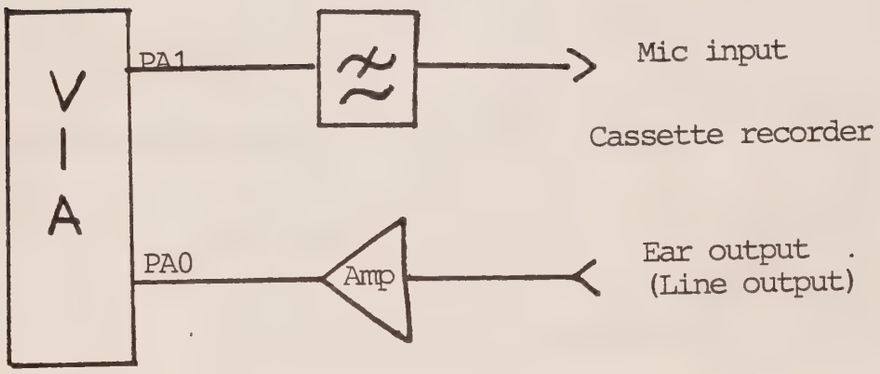


Fig 3.

APPENDIX B.

GLOSSARY

The following word list contains the words which are added to the FORTH-79 standard set off words. Attributes are indicated by capitalized symbols:

C	The word may only used within a colon-definition
E	The word may not be used within a colon definition.
I	The word is IMMEDIATE and will be executed during compilation, unless special action is taken.
U	The word is a USER variable.

Word	Description
(RUN)	proc addr ---- Run time primitive for RUN. Starts the word with execution adress addr in process with adress proc.
>?TERM	---- addr U USER-variable. Contains the PFA of the currently used ?TERMINAL word (Ch.5) Contains the PFA of (?TERMINAL) as default.
>EMIT	---- addr U USER-variable. Contains the PFA of the currently used EMIT word (Ch.5) Contains the PFA of (EMIT) as default.
>KEY	---- addr U USER-Variable. Contains the PFA of the currently used KEY word. (Ch.5) Contains the PFA of (KEY) as default.

n bytes of private dictionary space, see PROCESS.

INTVEC

---- addr

Global variable containing the address of the interrupt handling process.

IOPORT

---- addr

In some systems used to give the adress of the ACIA port handling the terminal device assigned to this process.

KEEP

Used by external interrupt processes. Will reserve the next time slice immediately following execution for the interrupt process.

LINK-IN

addr ----

Used by process to link job at addr into the process chain.

PRIORITY

---- addr U

User variable contains priority of process.

PROCESS nnnn

n ---- (compiling)
---- addr (run-time)

Allocates dictionary and status table space for a new job. Example of use :

```
200 PROCESS NEWJOB
```

allocates 200 bytes of dictionary space to the new process NEWJOB. At runtime NEWJOB leaves the address of its user-table. Further process constructs run-able process at addr. This new process inherits all USER table variable values with exception of R0,S0,DP and TIB. The process is linked into the process chain and put into the HALT status.

MOVE nnnn ----- (not all versions)

Removes the process with the name "nnnn" from the process chain. Results in error ERR 0 if not found in dictionary or process list.

IME ---- addr

Global double precision variable containing the number of seconds since system reset(Start). addr contains the LSI and addr+2 the MSI.

N nnnn addr ---- I

Gives the process att addr the word "nnnn" for execution. This word must HALT the process before reaching a ;S (or (;S)).

VE n ----

Saves screen n on tape.

ND cccc..... addr ----

Sends a text line to the job at addr. Activats job. The job is put into HALT status after end-of-line is encountered. (See DEF-TASK)

T-PRIO n addr ----

Sets the relative priority of the process at addr to the value n.

GNAL addr ---- C

Releases semaphore variable at addr (non-shared incrementation by 1). See WAIT

OP addr ----

Puts process at addr into HALT status.

ITCH ----

Switches process without affecting current process status.

'STAT ----

Prints the Process chain. For each process the name and current status is indicated. Example:

```
        JOB    PRIO STATUS
1     SYSTEM   100    RUN  *
2     NEWJOB   100    HALT
```

The '*' indicates the current process.

'STEM ---- addr

System constant giving the address of the SYSTEM (= default process).

'RMINAL-JOB ---- addr U

USER-variable. True content indicates that the process is a terminal handling process.

'INK ---- addr U

USER-variable giving the current process-link field address. The contents of this location points to next address in process-chain.

'AIT addr ---- C

This word will check the contents of the semaphore variable at addr and proceed if this contents is non-zero. In this case the variable will be decremented by 1. Otherwise WAIT will suspend execution of the process until thee contents of the semaphore variable becomes non-zero.

)STAT ---- addr

Returns address of logical variable containing the status of the watchdog. A true value means watchdog on status.

TGGLE

Alters the status of the watchdog.

EDITOR WORDS

CLEAR n ----

Ereases (fill with blanks) scr n.
Makes scr n current screen.

n ----

Delete line n and hold content at PAD

n ----

Erease (fill with blanks) line n

n ----

Hold line n at PAD without removing it.

n ----

Insert line from PAD before line n.

n ----

List current screen.

cccc n ----

Insert text on line n ereasing previous content.

n ----

Replace line n with content of PAD.

n ----

Make new line n, moving old text including the old line n down one line.

XT cccc c ----

Put text at PAD with delimiter c. If the byte c is not encountered the whole line is moved to PAD.

APPENDIX C.

Error messages

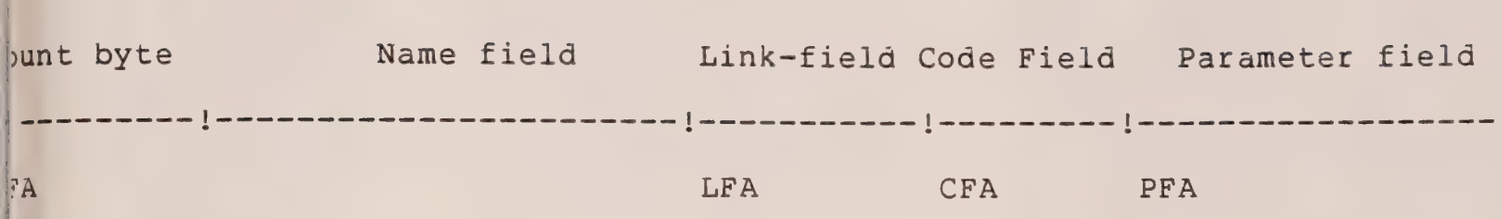
The FORTH error handler ERROR issues the messages listed below (if the ntents of WARNING is 0). In addition to that two numbers are left on the ack, indicating where in the input line (or block) the error occured. The top

mber indicates in which block the error occured. When this number is 0, the ror was found in the terminal input buffer (TIB). The second number indicates which position in the block (or TIB) the error was encountered. The numbers ven in the table below are the standard (FIG) error messages and should be oided for user error handling. Depending on the user variable TERMINAL-JOB, RROR will either halt the process, or control will resume to the interpreter p loop QUIT. The message numbers are decimal.

ror number	Interpretation
0	Word not found in dictionary
1	Empty stack (parameter stack underflow)
4	Definition is not unique (Warning only)
6	Block number out of range
7	Full stack (Parameter stack overflow)
17	Compilation only, use in definition
18	Execution only
19	Conditionals not paired
20	Definition not finished (missing ;)
21	In protected dictionary
22	Use only when loading
23	Character off current editing screen
24	Declare vocabulary

APPENDIX D.

Definition memory map



ount byte : Bit 7 Smudge bit, if not set: word is not valid (yet).
Bit 6 Precedence bit, if set: word is immediate.
Bit 5-0 Name field byte count (0-31).

ame field Variable length 7 bit ASCII. Last byte has bit 7 set.
ink field Address NFA of preceeding definition (2 bytes)
ode field Points to code to executed for this definition (2 bytes)
arameter field Variable length.

FORTH HANDY REFERENCE

1. Stack Manipulation

DUP	(n -> n n)	Duplicate top of stack
?DUP	(n1 -> n2)	Duplicate top element if it's not zero
DROP	(n ->)	Discard top of stack
SWAP	(n1 n2 -> n2 n1)	Exchange top elements of stack
ROT	(n1 n2 n3 -> n3 n1 n2)	Rotate third element to top
OVER	(n1 n2 -> n1 n2 n1)	Copy second element to top
PICK	(n1 -> n2)	Copy n1-th element of stack to top
ROLL	(n1 ->)	Rotate n-th element to top of stack
DEPTH	(-> n)	Count the number of stack items
SP!	(->)	Reset data (parameter) stack
>R	(n ->)	Move top of stack to top of return stack
R>	(-> n)	Move top of return stack to top of stack
R@	(-> n)	Copy top of return stack to top of stack
RP!	(->)	Reset return stack
S0	(-> addr)	User variable, contains start of stack.
R0	(-> addr)	User variable, contains start of return stack

2. Comparison

<	(n1 n2 -> f)	True if n1 less than n2
>	(n1 n2 -> f)	True if n1 greater than n2
=	(n1 n2 -> f)	True if n1 equal to n2
0<	(n1 -> f)	True if n1 negative
0>	(n1 -> f)	True if n1 positive
0=	(n1 -> f)	True if n1 equal to zero
D<	(d1 d2 -> f)	True if d1 less than d2 (double prec.)
U<	(u1 u2 -> f)	True if u1 less than u2 (unsigned)
NOT	(f -> f)	Reverse truth value (<=> 0=)

3. Arithmetic and logical

+	(n1 n2 -> n)	Add n = n1 + n2
D+	(d1 d2 -> d)	Double precision add
-	(n1 n2 -> n)	Subtract n= n1-n2
1+	(n -> n+1)	Add 1 to top element
2+	(n -> n+2)	Add 2 to top element
1-	(n -> n-1)	Subtract 1 from top element
2-	(n -> n-2)	Subtract 2 from top element
*	(n1 n2 -> n)	Multiply n = n1*n2
/	(n1 n2 -> n)	Divide n = n1/n2 (integer quotient)
2*	(n -> 2n)	Multiply top element by 2
2/	(n -> n/2)	Divide top element by 2
MOD	(n1 n2 -> rem)	Divide giving remainder from n1/n2
/MOD	(n1 n2 -> rem quot)	Divide giving remainder & quotient
*/MOD	(n1 n2 n3 -> rem quot)	Multiply then divide using double prec.
*/	(n1 n2 n3 -> quot)	As */MOD giving quotient only
U*	(un1 un2 -> ud)	Multiply unsigned
U/MOD	(ud un -> urem uquot)	Divide double number unsigned
MAX	(n1 n2 -> n)	Leave greater of two numbers
MIN	(n1 n2 -> n)	Leave lesser of two numbers
ABS	(n -> n)	Absolute value
NEGATE	(n -> -n)	Leave two's complement
DNEGATE	(d -> -d)	Leave two's complement (double prec.)
AND	(n1 n2 -> n)	Bitwise logical AND
OR	(n1 n2 -> n)	Bitwise logical OR
XOR	(n1 n2 -> n)	Bitwise logical exclusive OR.

4. Memory

@	(addr -> n)	Replace address by number at address
!	(n addr ->)	Store n at addr
C@	(addr -> byte)	Replace address with byte at addr.
C!	(byte addr ->)	Store byte at addr.
?	(addr ->)	Display number at addr
+!	(n addr ->)	Add n to contents at addr
CMOVE	(addr1 addr2 n ->)	Move n bytes starting at addr1 to addr2
FILL	(addr1 n byte ->)	Fill memory starting at addr1 with n bytes
BLANK	(addr1 n ->)	Fill memory starting at addr1 with Spaces
ERASE	(addr1 n ->)	Fill memory starting at addr1 with zeros

5. Control structures

DO	(end+1 start ->)	Set up loop with index range
LOOP	(->)	End of loop , add 1 to loop index
+LOOP	(n ->)	End of loop , add n to loop index
I	(-> index)	Place current loop index at top of stack
J	(-> index)	Place loop index of outer loop on stack
LEAVE	(->)	Terminate loop at next LOOP or +LOOP
IF	(f ->)	If true keep executing else goto ELSE or THEN
THEN	(->)	End of condition clause
ELSE	(->)	Start of else clause
ENDIF	(->)	Equivalent to THEN
BEGIN	(->)	Start of indeterminate loop
WHILE	(f ->)	Keep looping if true
UNTIL	(f ->)	End of loop if true
REPEAT	(->)	End of loop use with WHILE
END	(->)	Equivalent to REPEAT
AGAIN	(->)	Keep looping.
EXIT	(->)	Terminate execution of colon defintion.
EXECUTE	(cfa ->)	Execute word with code field addr on stack.

5. Terminal I/O

EMIT	(byte ->)	Type ASCII value from stack
KEY	(-> byte)	Get ASCII value from terminal
?TERMINAL	(-> f)	True if character available from terminal
(EMIT)	(byte ->)	Standard terminal EMIT word
(KEY)	(-> byte)	Standard KEY word
(?TERMINAL)	(-> f)	Standard ?TERMINAL word
>EMIT	(-> addr)	User variable containing pfa of EMIT word
>KEY	(-> addr)	User variable containing pfa of KEY word
>?TERM	(-> addr)	User variable containing pfa of ?TERMINAL
TYPE	(addr n ->)	Type string of n characters beginning at addr
COUNT	(addr -> addr+1 n)	Get count and start address of string at addr
-TRAILING	(addr n1 -> addr n2)	Delete trailing spaces of string
SPACE	(->)	Type a space
SPACES	(n ->)	Type n spaces
CR	(->)	Type a carriage return and line feed
EXPECT	(n addr ->)	Read n chars from terminal to memory at addr
QUERY	(->)	Get one line from terminal, put it at (TIB)
TIB	(-> addr)	User variable, contains addr of terminal buffer
>IN	(-> addr)	User variable, contains offset into TIB
OUT	(-> addr)	User variable, number of chars printed by EMIT.
BLK	(-> addr)	User variable, contains current block number.
WORD	(char -> addr)	Read next word from input. Delimiter: char,null
COLUMNS	(-> addr)	User variable, number of chars/line

6. Number Conversion

BASE	(-> addr)	User variable,contains base for num. conv.
DECIMAL	(->)	Sets base to 10
HEX	(->)	Sets base to 16
.	(n ->)	Print number with one trailing blank and sign.
.R	(n1 n2 ->)	Print n1 rigth adjusted using n2 print pos.
U.	(un ->)	Print unsigned number
D.	(d ->)	Print double number with sign
D.R	(d n ->)	Print double number rigth adjusted using n pos.
CONVERT	(d1 addr1 -> d2 addr2)	Convert string at addr1 and add to d1.
<#	(->)	Start numeric output
#	(ud1 -> ud2)	Convert next digit of ud1 and add to string.
#S	(ud1 -> 0 0)	Convert rest of digits in ud1.
HOLD	(char ->)	Add char to output string
SIGN	(n ->)	Add minus sign to output string if n<0
#>	(d -> addr n)	Drop d and get addr and count for TYPE
S->D	(n -> d)	Converts single to signed double number.

7. Mass storage

LIST	(n ->)	List screen n and set SCR to n
LOAD	(n ->)	Interpret from screen n
SCR	(-> addr)	User variable, contains current screen number
BLOCK	(n -> addr)	Leave address of blockbuffer containing block n
SAVE	(n ->)	Save screen n on tape.
GET	(->)	Get screen from tape.

8. Defining words

:		(->)	Begin colon definition
;		(->)	End of colon definition
CREATE	xxx	(->)	Create two byte variable xxx:
	xxx	(-> addr)	xxx returns address when executed
CONSTANT	xxx	(n ->)	Create constant with value n
	xxx	(-> n)	xxx returns value n
VARIABLE	xxx	(->)	Same as CREATE
	xxx	(-> addr)	
USER	xxx	(n ->)	Create user variable, process table offset n.
	xxx	(-> addr)	Return address of user variable.
DOES>	does:	(-> addr)	Used to define runtime behavior of word.

9. Vocabularies

VOCABULARY xxx	(->)	Create vocabulary xxx
xxx	(->)	Make vocabulary xxx context vocabulary.
DEFINITIONS	(->)	Make context vocabulary current vocabulary
FORTH	(->)	Main vocabulary.
' xxx	(-> addr)	Return pfa address of xxx on stack.
FIND xxx	(-> addr/0)	Return pfa address of xxx, if not found 0.
FORGET xxx	(->)	Forget all definitions made later than xxx.
FENCE	(-> addr)	User variable, contains lowest FORGET address.
LATEST	(-> addr)	Returns nfa address of most recent word.
WIDTH	(-> addr)	User variable, number of sign. chars in names.

10. Compiler

DP	(-> addr)	User variable, contains next free location.
HERE	(-> addr)	Gets contents of DP on stack.
,	(n ->)	Compile a number into dictionary at HERE
ALLOT	(n ->)	Add n bytes to parameter field of last word.
IMMEDIATE	(->)	Marks most recent word immediate
SMUDGE	(->)	Smudges most recent word.
."	(->)	Print or compile string terminate by "
LITERAL	(n ->)	If compiling, save n in dictionary.
DLITERAL	(d ->)	If compiling, save d in dictionary.
STATE	(-> addr)	User variable, contains true value if compiling
COMPILE	(->)	Compile address of the next non-immediate word.
[COMPILE]	(->)	Compile the next word regardless of status.
[	(->)	Stop compiling and begin executing.
]	(->)	Stop executing and resume compiling.

11. Editor

L	(->)	List current screen.
PL text	(n ->)	Insert text on line n.
HL	(n ->)	Hold line n at PAD.
DL	(n ->)	Delete line n holding the contents at PAD.
SL	(n ->)	Insert empty line at line n.
EL	(n ->)	Erease line n.
RL	(n ->)	Replace line n with contents of PAD.
IL	(n ->)	Put contents of PAD on new line n.
.PAD	(->)	Type contents of PAD.
TEXT xx	(ch ->)	Get text xx from input line to ch, store at PAD
(LINE)	(n scr -> addr)	Find addr of line n on screen scr.
.LINE	(n scr ->)	Type line n on screen n.

12. Multitasking

```
SEND     ( addr -> )          Send rest of line to process at addr.
RUN xxx  ( addr -> )          Let process at addr execute xxx
CONTINUE ( addr -> )          Let process at addr resume execution.
STOP     ( addr -> )          Halt process at addr
HALT     ( -> )               Halt executing process.
SWITCH   ( -> )               Surrender current time slice.
KEEP     ( -> )               Keep current time slice.
SIGNAL   ( addr -> )          Signal on semaphore variable at addr
WAIT     ( addr -> )          Wait on semaphore variable at addr
RSTPRO   ( addr -> )          Reinitializes process at addr.
PROCESS xxx      ( n -> )     Create process xxx with n bytes of dictionary.
        xxx      ( -> addr)   Returns addr of process.
INTPROCESS xxx   ( n -> )     Create interrupt process xxx
           xxx   ( -> addr )  Returns addr of process.
REMOVE xxx       ( -> )       Remove all processes created later than xxx.
PRIORITY         ( -> addr )  User variable, contains priority of process.
SET-PRIO         (n addr -> ) Set priority of process at addr to n.
TERMINAL-JOB     ( -> addr )  User variable, true if terminal handling.
SYSTAT   ( -> )               Print list of processes.
```

13. Error handling

```
ABORT    ( -> )               Warm restart of system, writes message.
(ABORT)  ( -> )               Execute word in >ABORT. Default = ABORT.
>ABORT   ( -> addr )          User variable. Contains pfa of error handler.
QUIT     ( -> )               Warm restart of system, no message written.
MESSAGE  ( n -> )             Issue error message n and continue execution.
ERROR    ( n -> )             Issue error message n and QUIT or (ABORT).
?ERROR   ( n f -> )           If f is true issue error message n and QUIT.
?STACK   ( -> )               Execute ERROR if stack out of bounds.
?FREE    ( -> )               Execute ERROR if <200 bytes of memory left.
?EXEC    ( -> )               Execute ERROR if not in execute state.
?COMP    ( -> )               Execute ERROR if not in compile state.
?PAIRS   ( n1 n2 -> )         Execute ERROR if n1 not equal to n2.
?LOADING ( -> )               Execute error if not interpreting from screen.
!CSP     ( -> )               Save current stack position.
?CSP     ( -> )               Execute error if stack pos. changed since !CSP.
 ARNING  ( -> addr )          User variable. <0 Error = (ABORT).
```

14. Miscellaneous

```
COLD     ( -> )               Cold restart = Reset. Dictionary is destroyed.
DUMP     ( addr  > )          Dump 256 bytes of memory starting at addr.
VLIST    ( -> )               Print the context vocabulary.
(        ( -> )               Begin comment terminated with )
PAD      ( -> addr )          Leave address of scratch area.
```